Vol. 22 No. 5　　情報処理学会論文誌　　Sep. 1981

スタック・マシンのための最適コード生成のアルゴリズム†

中田育男††

かぎられた長さの演算用スタック・レジスタを持つ計算機のための最適コードを生成する簡単なアルゴリズムを提案している．そのアルゴリズムは算術式を解析木に変換してからコード生成を行うものである．共通部分式の利用を考えないものはすでに一般的には解かれているが，ここではそれを，まず(1)分かりやすく，簡単なアルゴリズムとして表現し，さらに(2)交換命令，関数呼出し，共通部分式などのある場合に拡張している．

1. はじめに

FORTRAN などのプログラム言語の算術式の評価順序を，レジスタと主記憶との間のデータ転送が最小になるようにするアルゴリズムは，いくつか研究されてきた[1)~6)]．共通部分式の再利用を考慮しない範囲ではこの問題は解決されているが，共通部分式まで考慮すると難しい問題となる[7)]．

ここでは，まずかぎられた長さのスタック・レジスタを持つ場合の，共通部分式を考慮しない一つのアルゴリズムを提案する．この問題は文献 6) ですでに一般的に解かれている．今，またそれをとりあげる理由は以下のようなものである．

i）文献 6) は基本的な命令をそなえたスタック・マシンに対して完全なアルゴリズムを与えている．ただし，イ）後に述べるような付加的な命令は考慮していない．ロ）共通部分式を考慮していない．ハ）最適なコードを生成することの証明に重点が置かれている，演算子はすべて多数項演算子としている（このことは勿論，より一般化しているという意味があるのではあるが），などのため必ずしも読みやすくはない．

ii）最近，Am 9511 A (LSI) とか HITAC E 600 など，かぎられた長さの演算用スタック・レジスタを持つものが出ている．それらにすぐ使える，分かりやすい，簡単なアルゴリズムがあるとよい．

iii）スタックの先頭二つを入れかえる命令などの付加的な命令がある場合とか，関数呼出しがある場合など実用上の問題もつけ加える．

iv）このようなアルゴリズムは IBM/370 のようなレジスタ方式の場合よりも重要である．なぜなら，370 のような場合，どんな順序で式を評価していっても，レジスタに空きがなくなったら，適当にどれか一つのレジスタの内容を退避することで，最適とは言えなくてもそれほど問題なく計算できる．しかし，スタック・レジスタが計算の途中で足りなくなったら問題である．

最後に，共通部分式を利用するための一つのアルゴリズムを提案する．

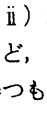
2. 基本アルゴリズム

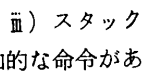
アルゴリズムの概要は，イ）算術式の解析木 (parse tree) の各節 (node) にラベルを付ける．ラベルはその節から始まる部分木を計算するのに必要なスタック・レジスタの数を示す．ロ）解析木を上からたどってコードを生成する．というものである．ラベルはすでにできている解析木に付けていくことにしてもよいが，解析木を作りながら付けていくことも考えられる．以下では N をスタック・レジスタの長さ（個数）とする．

イ）ラベル付けのアルゴリズム

節 a のラベルを l(a) とする

i）節 a が変数（または定数）なら l(a)=1

ii）節 a が  の形のとき l(a)=l(b)

iii）節 a が  のとき l(a)=max (l(b), l(c)+1) とするのであるが，その前に，l(c)=N なら c のコードを生成する．すなわち

GEN(c) ……次項の GEN でコード生成

〔pop w〕……結果を作業用番地wに格納するコードを生成．ここで〔α〕はコード α を生成することを意味する．

† Compiling Algorithms for Stack Machines by IKUO NAKATA (Institute of Information Sciences and Electronics, University of Tsukuba).

†† 筑波大学電子情報工学系

を行い，c を w で置きかえ $l(c)=l(w)=1$ とする．

ロ）コード生成のアルゴリズム GEN (a)

i）節 a が変数（または定数，作業用番地）のとき

〔push a〕

ii）節 a が a の形のとき（a が単項演算子）

b

GEN (b)

〔op a〕……op a は演算 a を行う命令コード

iii）節 a が a の形のとき

b　c

GEN (b)

GEN (c)

〔op a〕

ハ）全体のアルゴリズム

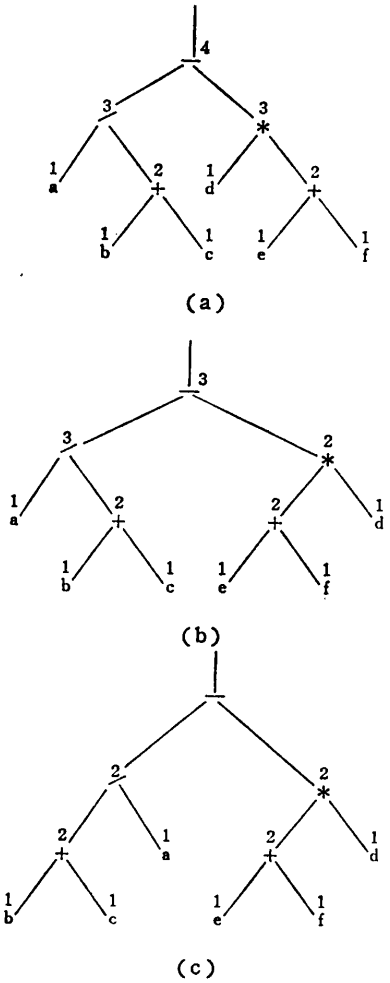


図 1 a/(b+c)−d∗(e+f) のラベル付解析木
Fig. 1 Labeled parse tree of a/(b+c)−d∗(e+f).

まずイ）によって解析木にラベルを付け（でき上がっている解析木に付けてもよいし，解析木を作りながら付けてもよい），次に GEN(解析木の根)を実行する．

例として

$$a/(b+c)-d*(e+f) \qquad (1)$$

をとりあげる．

以下では記述を短くするために push a を a，pop a を $\bar{a}$，op a を a と書く．(1)式のラベル付き解析木を図 1 の（a）に示す ($N\geq 4$)．それから得られるコードは

a b c + / d e f + ∗ −

となる．$N=3$ とすると "∗" の節のコードを先に生成することになり

d e f + ∗ $\bar{w}$ a b c + / w −

となる．$N=2$ とすると

b c + $\bar{w}_1$ e f + $\bar{w}_2$ d w_2 ∗ $\bar{w}_2$ a w_1 / w_2 −

となる．

上記のアルゴリズムでは "+" や "∗" が可換演算子であることを利用していない．それを利用するためにはイ）の iii）を次のように変更すればよい．

iii)′ 節 a が a の形のとき

b　c

1）$l(b)<l(c)$ で a が可換演算子なら b と c を入れかえ（もとの b(c) を新たな c(b) とし）

2）$l(c)=N$ なら

GEN (c)

〔pop w〕

を行い，c を w で置きかえ $l(c)=l(w)=1$ とし

3）$l(a)=\max(l(b), l(c)+1)$ とする．

このアルゴリズムで式(1)から得られるラベル付き解析木は図 1 の(b)となり ($N\geq 3$)，得られるコードは $N\geq 3$ のとき

a b c + / e f + d ∗ −

となる．$N=2$ のときは "+" (b+c) の節と "∗" の節のコードを先に生成することになり，

b c + $\bar{w}_1$ e f + d ∗ $\bar{w}_2$ a w_1 / w_2 −

となる．

以上のアルゴリズムは文献 6) のそれを簡単にしたものである．

3. 実用上の考慮

3.1 交換 (exchange) 命令がある場合

1 の ii）にあげた Am 9511 A や E 600 には交換命

令がある．それはスタックの先頭の二つを入れかえる命令である．これを利用して効果があるのは，たとえばイ）のⅲ)′で，$l(b)<l(c)=N$ でaが非可換演算子の場合で，このとき，bとcとを入れかえて

GEN (c)
GEN (b)
〔ex〕　……ex は交換命令
〔op a〕

とすればⅲ)′の 2) の pop w は不要になる．これを実現するためには ⅲ)′の 1) と 2) の間に次の 1′) を追加し，

1′) $l(b)<l(c)=N$ で a が非可換演算子なら b と c を入れかえ，a を a′ に変更し，

ロ）のⅲ）の〔op a〕を

a が a_1′ の形なら
　〔ex〕
　〔op a_1〕
そうでなければ
　〔op a〕

と変更すればよい．実用的にはこれでも使いものになるアルゴリズムかもしれないが，このアルゴリズムでは ex 命令を最適に利用するとはかぎらない．たとえば $l(b)=l(c)=N$ のとき節 c の下の方で入れかえを行えば $l(c)<N$ となるかもしれない．そこで，$l(b)<l(c)$ なら必ず入れかえを行って $l(a)$ をできるだけ小さくすると，今度はレジスタに余裕があるのに ex 命令が出てしまうことがある．それをふせぐためにコード生成の際，余分となった入れかえをもとに戻すことにする．以上の考察から以下のアルゴリズムが得られる．

イ)′ ラベル付けのアルゴリズム

節 a のラベルを $l(a)$ とする．

ⅰ）節 a が演算数（オペランド）なら $l(a)=1$

ⅱ）節 a が
　a
　b
の形のとき $l(a)=l(b)$

ⅲ）節 a が
　a
b　c
の形のとき

1) $l(b)<l(c)$ なら b と c を入れかえ，a が非可換演算子なら a を a′ に変更し，

2) $l(c)=N$ なら
　GEN (c, 0)
　〔pop w〕
　を行い，c を w で置きかえ（$l(c)=l(w)=1$ となる）

3) $l(a)=\max(l(b), l(c)+1)$ とする

ロ)′ コード生成のアルゴリズム GEN (a, k)

k はスタック・レジスタの余裕を示す数，最初は $k=N-l(a)$,

ⅰ）節 a が演算数のとき
　〔push a〕

ⅱ）節 a が
　a
　b
の形のとき
　GEN (b, k)
　〔op a〕

ⅲ）節 a が
　a
b　c
の形のとき

1）a が a_1′ の形でなければ

1.1) GEN (b, $k+l(a)-l(b)$)
　GEN (c, $k+l(a)-l(c)-1$)
　〔op a〕

2）a が a_1′ の形なら

$k\geq 1$ のとき b と c を入れかえ，a_1′ を a_1 に戻して上の 1.1) を行う．すなわち

GEN (c, $k+l(b)-l(c)$)……$k-1+\{l(b)+1\}-l(c)$
　$=k+l(b)-l(c)$
GEN (b, $k-1$)
〔op a_1〕

を行うことになる．

$k=0$ ならそのままで 1.1) を行う．ただし，そこで〔op a〕は〔ex〕〔op a_1〕を意味する．

本節の最初に述べた簡単なアルゴリズムによって，(1) 式から得られるラベル付解析木は，$N=2$ なら図1の(c)であり，$N=3$ なら(b)である．$N=2$ のときのコードは

e f+d*w̄₁ b c+a ex/w₁−

となる．この結果は最後のアルゴリズムから得られるものと同じである．両者のアルゴリズムの違いを示す例として

(a−b*c)/(d−e*f)　　(2)

をとりあげる．$N=3$ のとき前者のアルゴリズムで得られるラベル付解析木は**図 2**の(a)となる，したがって pop（主記憶へ格納）命令が一つ必要になる．後者のアルゴリズムで得られるものは 図 2 の(b)であり，コードの生成は

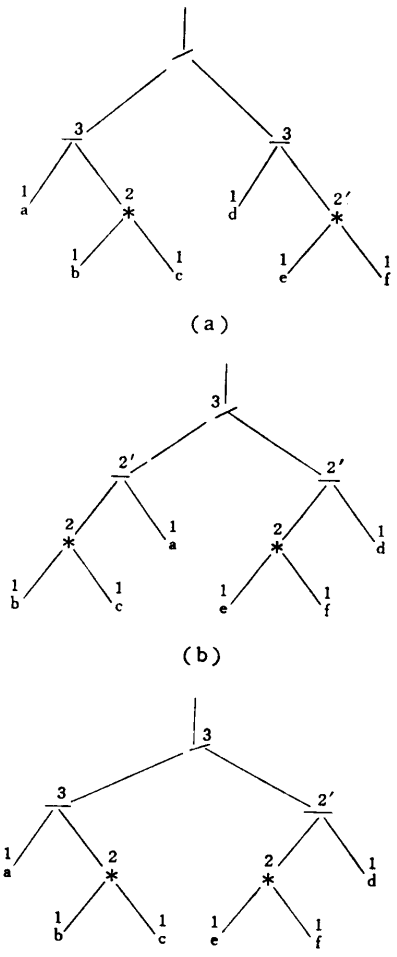


図 2 (a−b∗c)/(d−e∗f) のラベル付解析木
Fig. 2 Labeled parse tree of (a−b∗c)/(d−e∗f).

```
GEN (/, 0)
  GEN (−′, 1)
    GEN (a, 2)        ……push a
    GEN (∗, 0)
      GEN (b, 1)      ……push b
      GEN (c, 0)      ……push c
      〔op ∗〕           op  ∗
    〔op −〕           ……op  −
  GEN (−′, 0)
    GEN (∗, 0)
      GEN (e, 1)      ……push e
      GEN (f, 0)      ……push f
      〔op ∗〕           op  ∗
    GEN (d, 0)          push d
    〔op −′〕          ……ex
                        op  −
  〔op /〕             ……op  /
```

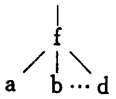
となり，結果として

a b c∗−e f∗d ex−/

が得られる．これは 図 2 の(b)の解析木を 図 2 の(c)の解析木に戻してからコード生成をした結果とも考えられる．

3.2 関数呼出し

式の中に関数参照があると，そこで関数呼出しが起る．関数参照 f(a, b, ……, d) の解析木は



の形とするのが普通である．このとき，f の関数サブルーチンの中で必要とするレジスタの数を R として，関数参照の節 f のラベルを R とすればよい[5),6),8)]ようにも思えるが，この式のコンパイル時には R が分からないかもしれないし，その関数のコンパイル時にはスタックがどこまで使われた状態で呼び出されるか分からない．そこで，一般には関数の入口ですべてのレジスタの内容を退避し，出口で回復するとするのが多いようであるが，それでは無駄が多い．その無駄をなくす一つの方法は，式の計算の途中で関数呼出しをするのでなく，関数呼出しを先にやってしまってから，式全体の計算をするものである．関数参照が入れ子になっている場合はもちろん内側のものから先に計算するようにする．

関数呼出しの結果の値は作業用番地に入れられて戻るという約束になっているシステムの場合は，関数呼出しの節をその作業用番地で置きかえればよい．

結果の値がスタックの先頭にある状態で戻る約束になっている場合に，それを最適に利用するのは簡単ではない．関数呼出しが複数個ある場合は，前の関数呼出しの結果，あるいはそれを使って演算した結果，を主記憶に格納してから次の関数呼出しを行うことになるので，この問題は，最後の関数呼出しの結果だけがスタックの先頭にある状態で考えればよい．すなわち，問題は，解析木の中の一つの葉だけがスタックの先頭にあるときの最適コードの求め方になる．これは簡単には解けない．必ずしも最適コードは求められないが，一つの現実的な方法としては次のようなものが考えられる．

上記の特別な葉を付けている節に関してはイ)，のⅲ)のａ）の左右の入れかえを，必要とあらばラベルに関係なく，行って，その葉が解析木の最左端の葉になるようにしていく．その途中で右側のラベルの値が N

になったらその部分木（その左端の葉が問題の葉）のコード生成をすることなど，それ以外は前記のアルゴリズムのままとする．このようにすることによってex 命令が増えるかもしれないが，主記憶との間の退避／回復が減ればその方が速い．

そのアルゴリズムは次のようになる．そこで $s(a)$ は

$$s(a)=\begin{cases}0：節\,a\,の左端の葉（a から左側の枝だけをたどって得られる葉）はスタックにある\\1：それ以外\end{cases}$$

を意味する．

イ)″ ラベル付けのアルゴリズム

節 a のラベルを $l(a)$ とする

i) 節 a が演算数（オペランド）なら $l(a)=1$ とし a がスタックにあるもの（iv)の a_0）なら $s(a)=0$，そうでなければ $s(a)=1$ とする

ii) 節 a が $\overset{|}{a}\overset{|}{b}$ の形（単項演算子）のとき $l(a)=l(b)$, $s(a)=s(b)$ とする

iii) 節 a が $\overset{|}{a}$ の形（2項演算子）のとき

$$\begin{array}{ccc}&|&\\b&&c\end{array}$$

1) $l(b)\cdot s(c)<l(c)\cdot s(b)$ なら b と c を入れかえ，そのとき a が非可換演算子なら a を a′ に変更し，

2) $s(a)=s(b)$ とし

3) $l(c)=N$ なら

GEN (c, 0)

〔pop w〕

を行い，c を w で置きかえ i) を行う（$l(c)=l(w)=1$, $s(w)=1$）

4) $l(a)=\max(l(b), l(c)+1)$ とする

iv) 節 a が関数参照のとき

a の各実引数 b のコードを生成し (GEN(b, $N-l(b)$), 〔pop w〕, を行う．$s(b)=0$ なるものがあればそれを先にする)，関数呼出しのコード (calling sequence) を生成し，a を「スタックにある」しるし a_0 で置きかえ上記 i) を行う．

ロ)″ コード生成のアルゴリズム GEN (a, k)

i) 節 a が演算数で $s(a)=1$ のとき

〔push a〕

($s(a)=0$ なら何も出力しない)

ii), iii) ロ)′ のそれと同じ．ただし iii) の 2) において $k\geq 1$ を $k\cdot s(b)\geq 1$, $k=0$ を $k\cdot s(b)=0$ で置きかえる．

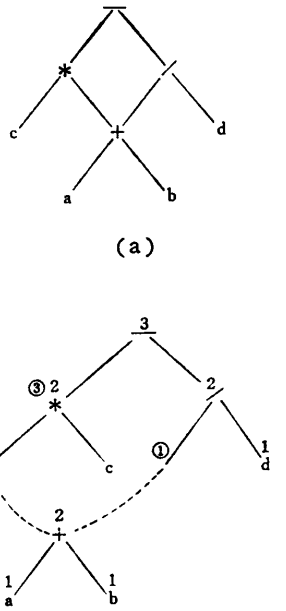


図 3　c∗(a+b)−(a+b)/d の DAG 表現とラベル

Fig. 3　DAG representation of c∗(a+b)−(a+b)/d.

ハ)″ 全体のアルゴリズム

まずイ)″ によってラベル付き解析木を作り，次に解析木の根 r に対して GEN(r, $N-l(r)$) を行う．

3.3 共通部分式

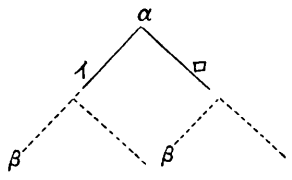
共通部分式を最適に利用する簡単なアルゴリズムはないであろうが，共通部分式を適当に利用する比較的簡単なアルゴリズムを考えることはできる．たとえば

c∗(a+b)−(a+b)/d

は a+b を共通部分式として持つ．そのことを認識して得られる解析木は図 3 の(a)のようになる．これは DAG (directed acyclic graph) と呼ばれる形である[8]．この DAG を図 3 の(b)のように変形してみる．これから次のコードが得られる．

a b+copy c∗ex d/−

ここで copy とはスタックの先頭の値と同じ値のもの (copy したもの) を push することを意味する．すなわちスタックの先頭部分には同じものが二つ並ぶことになる．一般には下図のように



α の左の枝イの左端と，右の枝ロの左端とが同じ部分式 β であったら，イの枝の β のコードの直後に copy 命令，イの枝のコードの直後に ex 命令（この命令でスタックの先頭に β の値がくる）を付ければよい．ロの枝の β のコードは必要ない．イの枝が β だけであったら ex 命令も省略できる．さらに α 自身が親の右側の部分木であればイの β のコードを省略できることもある．

上記のようにすると必要なレジスタの数が，前記のアルゴリズム イ）の l とは違ってくる．図3の(b)の③，①がそれを示す数である．そこで l のほかに次の l_l, l_r が必要になる．

l_l(a)：共通部分式を左端に持つaが「左」の枝として使われるときに必要なレジスタの数

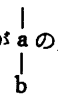
l_r(a)：上記の「左」を「右」にかえたもの

上記の図の β のような共通部分式をみつける部分のアルゴリズムとして以下のものが考えられる．そこでは共通部分式を指すポインタとして cs(a) を使っている．

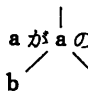
ニ）共通部分式を含んだ解析木のラベル付け

i）aがオペランドなら

l(a)=1, cs(a)=**nil** として iv）へ

ii）aが  の形なら

- l(a)=l(b), cs(a)=cs(b)
- cs(b)≠**nil** なら l_l(a)=l_l(b), l_r(a)=l_r(b)
- iv）へ

iii）aが  の形なら

1）cs(b)=**nil** で cs(c)≠**nil** なら b と c を入れかえて，そのとき a が非可換なら a を a′ に変更し，

2）cs(b)=cs(c)≠**nil** で l_l(b)≤N, l_r(c)<N なら（これが求めるもので，cs(b), cs(c) は共通部分式を指しているから，そこにこのことを記憶しておき，コード生成のとき，copy（命令 cs(b) のコードの直後）や空命令（cs(c) のコードとして）や，ex 命令（b のコードの直後）を生成できるようにする）

- l(a)=max (l_l(b), l_r(c)+1)
- l_l(a)=max (l_l(b), l_r(c)+2)

(l_r(c)+1+1 とするのは，a の計算の後で使う cs(b) の値の保持のレジスタが余分に必要だからである）

- l_r(a)=max (l_r(b), l_r(c)+1)
- cs(a)=cs(b)
- iv）へ

3）（上記2）以外のとき，cs(c)≠**nil** のときは，cs(b) と cs(c) は相異なる共通部分式である．その両方を生かそうとするアルゴリズムは複雑になるので，そのとき cs(c) の利用はあきらめることにする）

- l(a)=max (l(b), l(c)+1)
- cs(a)=cs(b)
- cs(b)≠**nil** のとき

  l_l(a)=max (l_l(b), l(c)+2)

  l_r(a)=max (l_r(b), l(c)+1)
- iv）へ

iv）aが共通部分式のとき

- l_l(a)=max (l(a), 2)
- l_r(a)=1
- cs(a)=a（a自身を指すポインタ）

例として

a/((a−b)*(a−c))

を考える．この DAG 表現にラベルを付けたものは**図4**となり，N≥4 のとき㋑と㋥，および㋥と㋭が利用できる共通部分式となるから，得られるコードは

a copy copy b−ex c−*/

となる．

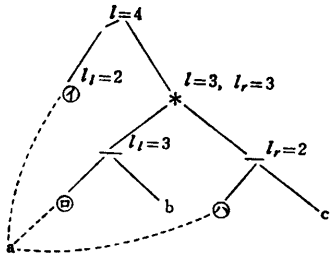


図4 a/((a−b)*(a−c)) の DAG 表現とラベル
Fig. 4 Labeled DAG of a/((a−b)*(a−c)).

上記のアルゴリズムで利用されないことになる共通部分式については，その値を作業用番地に格納して利用することとする（共通部分式が演算子を含まない場合はその必要はない）．

4. おわりに

かぎられた長さの演算用スタック・レジスタを持つ計算機のための最適コード（共通部分式については必ずしも最適コードとはならないかもしれないが）を生成する簡単なアルゴリズムを提案した．その基本アルゴリズムは Bruno[6] のそれを簡単にしたものであり，

さらに，交換 (exchange) 命令や複写 (copy) 命令を利用して，レジスタをより効率よく使うアルゴリズムを付加したものである．

スタック・マシンの場合，通常の複数個の汎用レジスタを持つマシンの場合と比べて，命令でレジスタ名を指定する必要がない，ハード/ソフトが簡単になる，といった利点があるが，演算に使われるレジスタはスタックの先頭のものと決っているので，演算の進め方の自由度が落ちる．そのため，たとえば同じ式を(途中結果を退避せずに)計算するのに必要なレジスタの個数が大きくなる可能性があるし，共通部分式の利用がしにくい．また1章iv) で述べたように，レジスタ最適利用のアルゴリズムがより重要となる．本アルゴリズムはそれらに対する一つの解となるものである．**3.1** 節のアルゴリズムによれば，同じ式の計算に必要なレジスタ数は，汎用レジスタ・マシンの場合と同じになる．**3.3** 節のアルゴリズムのように複写命令や交換命令を利用することによって，スタック・マシンでも共通部分式を，レジスタに置いたままで，活用することが可能になる．

本アルゴリズムは HITAC E 600 の FORTRAN コンパイラに，共通部分式に関するものを除いて，とり入れられている．共通部分式に関するものは時間的関係で最初の版にはとり入れられなかった．E 600 には交換命令も複写命令もある．スタック・レジスタの深さ（個数）N は5である．**3.1** 節のアルゴリズムによればラベルが増加する（＋1される）のは $l(b)=l(c)$ のときだけであるから，オペランドの個数が $2^5-1=31$ 以下の式は途中結果を退避 (pop) しないで演算することができる．

最後に，本アルゴリズムの必要性を示唆された，日立製作所ソフトウェア工場の堂免部長に深く感謝する．

参考文献

1) Anderson, J. P.: A note on some compiling algorithms, CACM, Vol. 7, No. 3, pp. 149–150 (1964).
2) Ershov, A. P.: On programming of arithmetic operations, CACM, Vol. 1, No. 8, pp. 3–6 (1958).
3) Nakata, I.: On compiling algorithms for arithmetic expressions. CACM. Vol. 10, No. 8, pp. 492–494 (1967).
4) Redziejowski, R. R. : On arithmetic expressions and trees, CACM, Vol. 12, No. 2, pp. 81–84 (1969).
5) Sethi, R. and Ullman, J. D.: The generation of optimal code for arithmetic expressions, JACM, Vol. 17, No. 4, pp. 715–728 (1970).
6) Bruno, J. and Lassagne, T.: The generation of optimal code for stack machines, JACM, Vol. 22, No. 3, pp. 382–396 (1975).
7) Aho, A. V. and Sethi, R.: How hard is compiler code generation, Lecture notes in Comp. Sci., No. 52 (Automata, Languages and Programming), Springer-Verlag, pp. 1–15 (1977).
8) Aho, A. V. and Ullmann, J. D.: Principles of Compiler Design, Addison-Wesley (1977).

(昭和55年11月17日受付)
(昭和56年2月19日採録)